Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model

Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, Samuel Thibault

Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model

Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, Samuel Thibault

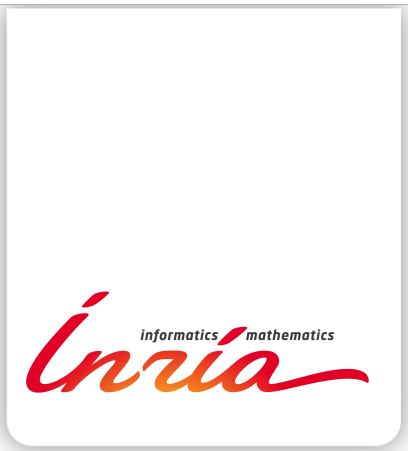
RESEARCH REPORT

N° 8927

June 2016

Project-Teams STORM and HiePACS

ISSN 0249-6399 ISRN INRIA/RR--8927--FR+ENG



Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model

Emmanuel Agullo, Olivier Aumage, Mathieu Faverge*, Nathalie Furmento†, Florent Pruvost, Marc Sergent‡, Samuel Thibault§

Project-Teams STORM and HiePACS

Research Report n° 8927 — June 2016 — 27 pages

Abstract: The emergence of accelerators as standard computing resources on supercomputers and the subsequent architectural complexity increase revived the need for high-level parallel programming paradigms. Sequential task-based programming model has been shown to efficiently meet this challenge on a single multicore node possibly enhanced with accelerators, which motivated its support in the OpenMP 4.0 standard. In this paper, we show that this paradigm can also be employed to achieve high performance on modern supercomputers composed of multiple such nodes, with extremely limited changes in the user code. To prove this claim, we have extended the StarPU runtime system with an advanced inter-node data management layer that supports this model by posting communications automatically. We illustrate our discussion with the task-based tile Cholesky algorithm that we implemented on top of this new runtime system layer. We show that it allows for very high productivity while achieving a performance competitive with both the pure Message Passing Interface (MPI)-based ScaLAPACK Cholesky reference implementation and the DPLASMA Cholesky code, which implements another (non sequential) task-based programming paradigm.

Key-words: runtime system, sequential task flow, OpenMP depend clause, heterogeneous computing, distributed computing, multicore, GPU, Cholesky factorization

* Bordeaux INP
† CNRS
‡ CEA
§ University of Bordeaux

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Obtenir de hautes performances sur les supercalculateurs avec un modèle de programmation par tâches séquentiel

Résumé : L'émergence d'accélérateurs comme ressources de calcul standard sur les supercalculateurs et la complexification associée des architectures ressuscite le besoin de paradigmes de programmation de haut niveau. La programmation par tâches séquentielle a montré qu'elle pouvait efficacement répondre à ce besoin sur un seul nœud de calcul multicœur possiblement dotée d'accélérateurs, ce qui a motivé son intégration dans le standard OpenMP 4.0. Dans ce papier, nous montrons que ce paradigme peut également être utilisé pour obtenir de hautes performances sur les supercalculateurs modernes composés de plusieurs de ces nœuds de calcul, tout en limitant à au maximum les changements dans le code utilisateur. Afin de prouver cette affirmation, nous avons ajouté au support d'exécution StarPU une couche avancée de gestion des données inter-nœuds qui supporte ce modèle en émettant automatiquement les communications. Nous illustrons notre discussion avec une factorisation de Cholesky tuilée en tâches que nous avons implémentée sur cette nouvelle couche de support exécutif. Nous montrons que cette méthode permet une grande productivité tout en permettant d'obtenir des performances compétitives tant avec l'implémentation de référence ScaLAPACK de Cholesky, qui est basée purement sur l'Interface de Passage de Message (MPI), qu'avec le code Cholesky de DPLASMA, qui implémente un autre modèle de programmation par tâches (non séquentiel).

Mots-clés : support d'exécution, flot de tâches séquentiel, clause OpenMP depend, calcul hétérogène, calcul distribué, multi-cœur, GPU, factorisation de Cholesky

1 Introduction

While low-level designs have long been the key for delivering reference high performance scientific codes, the ever growing hardware complexity of modern supercomputers led the High Performance Computing (HPC) community to consider high-level programming paradigms as solid alternatives for handling modern platforms. Because of the high level of productivity it delivers, the Sequential Task Flow (STF) paradigm – further introduced in Section 2.1 – is certainly the most popular of these candidates. On the one hand, many studies have shown that task-based numerical algorithms could compete against or even overpass state-of-the-art highly optimized low-level peers in areas as diverse as dense linear algebra [1, 2, 3, 4], sparse linear algebra [5, 6, 7], fast multipole methods [8, 9], $\mathcal{H}$-matrix computation [10] or stencil computation [11, 12], for quoting a few. On the other hand, task-based runtime systems have reached a high level of robustness, incurring very limited management overhead while enabling a high level of expressiveness as further discussed in Section 2.2.

The consequence is twofold. First, new scientific libraries based on the STF paradigm and relying on runtime systems have emerged. We may for instance quote the PLASMA [1], MAGMA [2], FLAME [3] and Chameleon [4] dense linear algebra libraries, the PaStiX [5] and QR_Mumps [6] sparse direct solvers, the Fast Multipole Method (FMM) ScalFMM [9] libraries. Second, the OpenMP Architecture Review Board has introduced constructs for independent tasks in the 3.0 and 3.1 revisions and dependent tasks in the 4.0 revision, which corresponds to a decisive step towards the standardization of the STF paradigm.

The aforementioned results showed the success of STF in exploiting a complex, modern, possibly heterogeneous node. However, because it requires to maintain a consistent view of the progress, the model has not been so far considered as a *scalable candidate* for exploiting an entire supercomputer. In the present article, we show that clever pruning techniques (Section 4.3.1) allow us to alleviate bottlenecks without penalizing the gain of productivity provided by the paradigm. Together with a careful design of communication (Section 4.3.2), allocation (Section 4.3.3) and submission (Section 4.3.4) policies, we show that this approach makes the STF model extremely competitive against both the native MPI and the Parameterized Task Graph (PTG) paradigms.

We carefully present the impact on performance and on the compactness of the different paradigms considered in this study. We illustrate our discussion with the tile Cholesky factorization algorithm [13, 1, 14] from the Chameleon solver [4] running on top of the StarPU runtime system [15]. The Cholesky factorization aims at factorizing Symmetric Positive Definite (SPD) matrices. We chose to illustrate our discussion with this routine as it is a simple algorithm (composed of only three nested loops as shown in Algorithm 1), and yet it is the reference factorization routine used in state-of-the-art libraries such as FLAME, LAPACK, ScaLAPACK, PLASMA, DPLASMA, or MAGMA for solving linear systems associated to SPD matrices. Tile algorithms [1], first implemented in the PLASMA library for multicore architectures, are now reference algorithms on parallel architectures and have been incorporated into the Chameleon, FLAME, Intel MKL and DPLASMA libraries. While PLASMA relies on the light-weight Quark [16] runtime system, Chameleon is a runtime-oblivious extension of PLASMA designed for research perspectives that can run on top of many runtime systems, including Quark and StarPU. The contribution of this article is the study of the potential of the STF model for running at large scale on a parallel distributed supercomputer. We chose to extend the StarPU (read *PU) runtime system to illustrate this claim, in order to inherit from its ability to abstract the hardware architecture (CPU, GPU, ...), hence being able to exploit heterogeneous supercomputers.

The rest of the paper is organized as follows. Section 2 presents task-based programming

Algorithm 1 Baseline tile Cholesky algorithm

```
for (k = 0; k < N; k++) do
  POTRF (A[k][k]);
  for (m = k+1; m < N; m++) do
    TRSM (A[k][k], A[m][k]);
  end for
  for (n = k+1; n < N; n++) do
    SYRK (A[n][k], A[n][n]);
    for (m = n+1; m < N; m++) do
      GEMM (A[m][k], A[n][k], A[m][n]);
    end for
  end for
end for
```

models (Section 2.1), runtime systems that support them in a distributed memory context (Section 2.2) and the baseline (non distributed) version of the StarPU runtime system (Section 2.3). Section 2.4 presents an extension of StarPU to support explicit MPI directives for exploiting distributed memory machines [17]. Section 3 builds on top of it to make communication requests implicit (first contribution), *i.e.* transparent to the programmer and automatically posted by the runtime system, provided that an initial data mapping is supplied by the programmer. Section 4 presents a detailed performance analysis together with the list of optimizations needed for efficiently supporting the STF model on modern supercomputers (second contribution) before concluding remarks are discussed in Section 5.

2 Background

2.1 Task-based programming models

Modern task-based runtime systems aim at abstracting the low-level details of the hardware architecture and enhance the portability of performance of the code designed on top of them. In many cases, this abstraction relies on a *directed acyclic graph (DAG) of tasks*. In this DAG, vertices represent the tasks to be executed, while edges represent the dependencies between them.

Each runtime system usually comes with its own API which includes one or multiple ways to encode the dependencies and their exhaustive listing would be out of the scope of this paper. However, we may consider that there are two main modes for encoding dependencies. The most natural method consists in declaring *explicit dependencies* between tasks. In spite of the simplicity of the concept, this approach may have a limited productivity in practice, as some algorithms may have dependencies that are cumbersome to express. Alternatively, dependencies originating from tasks accessing and modifying data may be *implicitly* computed by the runtime system, thanks to the *sequential consistency*. In this latter approach, tasks are submitted in sequence and the data they operate on are also declared.

Depending on the context, the programmer affinity and the portion of the algorithm to encode, different paradigms may be considered as natural and appropriate, and runtime systems often allow to combine them. Alternatively, one may rely on a well-defined, simple, uncluttered programming model in order to design a relatively more simple code, easier to maintain and benefit from properties provided by the model.

The main task-based programming model relying on *explicit dependencies* between tasks is certainly the *Parameterized Task Graph* (PTG) model [18]. In this model, tasks are not enumer-

Algorithm 2 TRSM kernel of the PTG tile Cholesky

```
TRSM(k, m)

// Execution space
k = 0   .. N-1
m = k+1 .. N-1

// Task Mapping
: A[m][k]

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? A[m][k]
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..N-1, m)
        -> A[m][k]

BODY
  trsm( A /* A[k][k] */,
        C /* A[m][k] */ );
END
```

ated but parameterized, and dependencies between tasks are explicit. For instance, Algorithm 2 shows how the TRSM task from the above tile Cholesky algorithm is encoded in this model. The task is parametrized with k and m indices. Its execution space is defined as the range of values that these indices can have (for instance k varies from 0 to $N-1$). Assuming that a data mapping has been defined separately and ahead of time (for all tiles and hence for tile $A[m][k]$ in particular), a task mapping is provided by assigning task $TRSM(k,m)$ onto the resource associated with a certain data ($A[m][k]$ here). Then, dependencies are explicitly stated. For instance, $TRSM(k,m)$ depends on $POTRF(k)$ (left arrow) and, conversely, $SYRK(k,m)$ may depend on it (right arrow). Correctly expressing all the dependencies is not trivial, as one can see in the pseudo-code. Furthermore, in order to handle distributed memory machines, the data-flow must also be provided. TRSM has to retrieve two data, referenced as A and C in its body (where the actual code must be provided). In the particular case of the dependency between $TRSM(k,m)$ and $POTRF(k)$, POTRF produces one single data (noted A in the pseudo-code of POTRF), which must be transferred to TRSM and referenced as A for future usage within the body. For a matter of conciseness, we do not report POTRF, SYRK and GEMM pseudo-codes here; they are similar and further details can be found in [19]. This encoding of the DAG allows for a low memory footprint and ensures limited complexity for parsing it while the problem size grows. Furthermore, since dependencies are explicit, the DAG can be naturally unrolled concurrently on different processes in a parallel distributed context, a key attribute for achieving scalability.

On the other hand, the *Sequential Task Flow* (STF) programming model consists in fully relying on `sequential consistency`, exclusively using implicit dependencies. This model allows the programmer to write a parallel code that preserves the structure of the sequential algorithm, enabling for a very high productivity. Indeed, writing an STF code consists of inserting a

Algorithm 3 STF tile Cholesky

```
for (k = 0; k < N; k++) do
  task_insert(&POTRF, RW, A[k][k], 0);
  for (m = k+1; m < N; m++) do
    task_insert(&TRSM, R, A[k][k], RW, A[m][k], 0);
  end for
  for (n = k+1; n < N; n++) do
    task_insert(&SYRK, R, A[n][k], RW, A[n][n], 0);
    for (m = n+1; m < N; m++) do
      task_insert(&GEMM, R, A[m][k], R, A[n][k],
        RW, A[m][n], 0);
    end for
  end for
end for
task_wait_for_all();
```

sequence of tasks through a non-blocking function call (which we call "task_insert") that delegates the *asynchronous*, *parallel* execution of the tasks to the runtime system. Upon submission, the runtime system adds a task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [20]. The task becomes ready for execution only when all its dependencies are satisfied. A scheduler is then in charge of mapping the execution of ready tasks to the available computational units. In this model, the sequential tile Cholesky algorithm can simply be ported to the STF model as proposed in Algorithm 3. The only differences with the original Algorithm 1 are the use of *task_insert* instead of direct function calls, and the addition of explicit data access mode (R or RW). The increasing importance of this programming model in the last few years led the OpenMP board to extend the standard to support the model through the *task* and *depend* clauses in the revision 4.0. This paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors, where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies.

In a parallel *distributed* context, maintaining the sequential consistency requires to maintain a global view of the graph. In the literature, different approaches have been proposed. ClusterSs [21] has a master-slave model where a master process is responsible for maintaining this global view and delegates actual numerical work to other processes. The authors showed that the extreme centralization of this model prevented it to achieve high performance at scale. On the contrary, in the extension of Quark proposed in [22], all processes fully unroll the DAG concurrently. Although requiring fewer synchronizations, the authors showed that the approach was also limited at scale, as each process is still required to fully unroll the DAG, and in spite of the lower number of synchronizations, that symbolic operation is performed redundantly and takes a significant amount of time, possibly higher than the time spent for numerical computations. In the present article, we study the potential of the STF paradigm for programming modern supercomputers relying on the StarPU runtime system (see Section 2.3). As in [22], all processes fully unroll the DAG concurrently. However, while in [22], the full unrolling of the DAG was a serious drawback for performance, we propose to alleviate this bottleneck by performing a clever pruning of the DAG traversal (see Section 4.3.1) to entirely eliminate irrelevant dependence edge instantiation, and we show that the STF model then becomes competitive against both the native MPI and PTG paradigms.

2.2 Short review of task-based runtime systems for distributed memory platforms

Several related approaches have been proposed within the community. StarSs is a suite of runtime systems developed at the Barcelona Supercomputing Center and supporting the STF model. Among them, ClusterSs [21] provides an STF support for parallel distributed memory machines with a master-slave interaction scheme. OmpSs [23] targets SMP, SMP-NUMA, GPU and cluster platforms. Master-slave schemes may suffer from scalability issues on large clusters due to the bottleneck constituted by the master, though OmpSs supports task nesting on cluster nodes, to reduce the pressure on the main master node. On the contrary, the extension of Quark for distributed memory machines proposed in the PhD thesis of YarKhan [22] at University of Tennessee Knoxville (UTK) relies on a decentralized approach to support the STF model. The present study extends this decentralized approach with the StarPU runtime system.

The PaRSEC runtime system [24, 2] developed at UTK supports the PTG model (as well as other Domain Specific Languages). Dependencies between tasks and data are expressed using a domain specific language named JDF, and compiled into a compact, parametric representation of the dependence graph. The size of the PTG representation does not depend on the size of the problem, but only on the number of different tasks used by the application, allowing for an efficient and highly scalable execution on large heterogeneous clusters. JDF also ensures support for irregular applications. Because dependencies are explicitly provided by the application, the model is extremely scalable: the runtime can explore the DAG from any local task to any neighbor task. PaRSEC exploits this property to ensure an excellent scalability. The DPLASMA library is the dense linear algebra library implemented on top of PaRSEC that originally motivated the design of this runtime. It is highly optimized and can be viewed as a reference implementation of a library on top of PaRSEC (although many other scientific codes have been implemented on top of it since then). DPLASMA/PaRSEC was the challenged code in [22] to assess the limits of the STF support for distributed memory platforms in the proposed extension of Quark. It will also be our reference code.

The SuperGlue [25] environment developed at the University of Uppsala provides a model based on data versioning instead of using a DAG of tasks. SuperGlue only implements a single scheduling algorithm: locality-aware work stealing. It is limited to single node, homogeneous multicore platforms. However it can be associated with the DuctTEiP [26] environment, to target homogeneous clusters.

The Legion [27] runtime system and its Regent compiler [28] provide a task-based environment supporting distributed memory platforms and GPUs, programmable at the Legion library level directly, or at a more abstract level through the Regent dedicated language and compiler. Legion enables the programmer to define *logical regions* similar to StarPU's registered data, and express dependencies between tasks and logical region accesses. In contrast to StarPU tasks, Legion tasks may spawn child tasks and wait for their completion. StarPU tasks must instead run to completion as a counterpart for enabling accurate performance model building.

The TBLAS environment [29], initially designed at UTK, is a task-based runtime system for clusters equipped with GPUs. It makes use of both the general purpose cores and the GPUs for computations. However, tasks as well as data are distributed to hosts and GPUs statically contrary to the approach we consider in the present paper where any task can run on any computational unit thanks to the abstraction of the STF model we consider.

Many other runtimes have been designed over the years. APC+ [30] and Qilin [31] only optimize scheduling for a single kind of computing kernel per application. ParalleX/HPX [11] supports both parallel and distributed memory platforms, but no detail is given about accelerator support. Most of them perform reactive/corrective load balancing using approaches such as

work stealing [32] or active monitoring [33, 34] while StarPU, used for this work, attempts to anticipatively map computations in a balanced manner using performance models.

PGAS languages such as UPC or XcalableMP provide distributed shared memory and are being extended to CUDA devices [35, 36]. The proposed interfaces are however mostly guided by the application, without dynamic scheduling, thus leaving most of the load balancing burden on the application programmer.

2.3 StarPU: a task-based runtime system for heterogeneous architectures

The ground for this work is the StarPU [15] runtime system, which deals with executing task graphs on a single heterogeneous node, composed of both regular CPU cores and accelerators such as GPUs.

Algorithm 4 Registration of elemental data

```
for (m = 0; m < N; m++) do
  for (n = 0; n < N; n++) do
    starpu_data_register(&Ahandles[m][n], A[m][n], ...);
  end for
end for
```

The principle of StarPU is that the application first *registers* its data buffers to StarPU, to get one *handle* per data buffer, which will be used to designate this data, as shown in Algorithm 4. This allows StarPU to freely transfer the content back and forth between accelerators and the main memory when it sees fit, without intervention from the application. For each operation to be performed by tasks (the GEMM, GEneral Matrix Multiplication, for instance), a *codelet* is defined to gather the various implementations: the `DGEMM` CPU implementation from MKL and the `cublasDgemm` GPU implementation from CUBLAS, for instance. A *task* is then simply a codelet applied on some handles. The StarPU runtime system can then freely choose when and on which CPU core or accelerator to execute the task, as well as when to trigger the required data transfers.

As a consequence, StarPU can optimize task scheduling by using state-of-the-art heuristics. Estimations of task completion time can typically be obtained by measuring them at runtime [37], or can be provided explicitly by the application. This allows StarPU to provide various scheduling heuristics ranging from basic strategies such as eager, or work stealing, to advanced strategies such as HEFT [38]. These can take into account both computation time and CPU–Accelerator device data transfer time to optimize the execution.

StarPU can further optimize these data transfers between the main memory and the accelerators memory. Since data registration makes StarPU responsible for managing data buffers locations, StarPU can keep track of which of the main memory and/or accelerators memory have a valid copy of a given data, using its distributed shared-memory manager. This allows for instance to replicate data over all accelerators which need them. By planning scheduling decisions in advance, StarPU can also start the required data transfers early, thus overlapping them with the currently running tasks. StarPU can also keep data in an accelerator as long as it is used repeatedly by tasks scheduled on it, thus avoiding duplicate transfers. When room must be made in the accelerator memory, on the contrary, it can also anticipatively evict unused data as well.

StarPU supports the STF model. The tile algorithm proposed in Algorithm 3 can be executed with StarPU, by simply replacing the *task_insert* call by *starpu_task_insert*, and the

Algorithm 5 Excerpt of MPI tile Cholesky algorithm using StarPU-MPI (panel only)

```
if ( myrank == Owner(A[k][k]) ) then
  starpu_task_insert(&POTRF_cl,
     STARPU_RW, Ahandles[k][k], 0);
  for (m = k+1; m < N; m++) do
    if ( myrank != Owner(A[m][k]) ) then
      starpu_mpi_isend_detached(Ahandles[k][k], ...);
    else
      starpu_task_insert(&TRSM_cl,
        STARPU_R, Ahandles[k][k],
        STARPU_RW, Ahandles[m][k], 0);
    end if
  end for
else
  for (m = k+1; m < N; m++) do
    if ( myrank == Owner(A[m][k]) ) then
      starpu_mpi_irecv_detached(Ahandles[k][k], ...);
      starpu_task_insert(&TRSM_cl,
        STARPU_R, Ahandles[k][k],
        STARPU_RW, Ahandles[m][k], 0);
    end if
  end for
end if
```

task_wait_for_all call by *starpu_task_wait_for_all*.

2.4 StarPU-MPI: Explicit message passing support for StarPU

While the StarPU runtime system [15] was intended to support the execution of a task-based code on a single node, we now present a mechanism proposed in [17], which allows to handle distributed memory platforms, named StarPU-MPI in the sequel. Provided the huge amount of existing MPI applications, one may indeed want to make it possible to accelerate these so that they can take full advantage of accelerators thanks to StarPU, while keeping the existing MPI support as it is. Instead of having a single instance of StarPU distributed over the entire cluster, the approach consists in having an instance of StarPU initialized on each MPI node and communicating with each other through StarPU-MPI. The flexibility of this hybrid programming model has also been illustrated in the case of MPI applications which call libraries written in OpenMP or TBB for instance.

Algorithm 5 shows how to write the tile Cholesky algorithm with this paradigm. For a matter of conciseness and clarity, we focus on the panel factorization only (a POTRF kernel and the subsequent TRSM kernels) and present a non optimized version. All function calls are asynchronous and allow for overlapping communications with task executions. The `for` loops thus complete quickly (they only submit requests to StarPU and StarPU-MPI) and the only synchronization point possibly occurs at the end of the application at the `starpu_task_wait_for_all` step, which ensures the completion of all submitted tasks. The `detached` calls perform *asynchronous* data transfers: for instance, the send requests will actually be posted to MPI only when the tasks which produce the data are finished, and tasks which depend on data coming from a reception request will also wait for the reception to complete.

While we presented a non optimized version of the code and restricted ourselves to the panel factorization only, one can observe that the code already differs from the STF paradigm. In the case of more complex algorithms (including a full and optimized version of the tile Cholesky factorization), this model may lead to error-prone and hard to maintain codes. In the sequel, we propose a programming model that sticks as much as possible to the STF paradigm by posting communications automatically, and allows for exploiting an entire modern supercomputer without the complexity of such a hybrid programming model.

3 Sequential task flow with implicit message passing

Just like explicit dependencies, explicitly specifying MPI communications is tedious and error-prone. We now propose to automatically infer those communications in order to maintain a compact STF code while exploiting a full heterogeneous supercomputer. The principle is the following. An initial data mapping over the MPI nodes is supplied by the application, and the sequence of tasks is identically submitted by the application on all MPI nodes. Each node can then unroll the whole application task graph, and automatically determines by itself which subset of tasks it should run (according to the data mapping), which MPI transfers it should initiate to resolve an out-going inter-node dependence edge, and which incoming MPI transfers it should expect resulting from its own inbound internode dependence edges. Indeed, an MPI send has to be automatically initiated when a local data is needed by a task running on an other MPI node, and an MPI receive has to be initiated when a local task needs a data which is mapped on another MPI node. Put another way, an MPI transfer is considered for each task graph edge between two tasks which are to be executed on different MPI nodes. Subsequently, no coherency synchronization is needed between nodes by construction, beyond the necessary user data transfers. Figure 1 illustrates those data transfers. A cache mechanism, described in Section 4.3.2, avoids duplicate communications. Moreover, since unrolling the whole task graph on each node can reveal costly at scale, we discuss in Section 4.3.1 how we refine the model to overcome this.

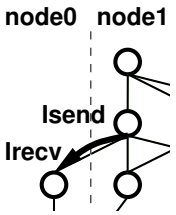


Figure 1: Example of communication inferred from task graph edge.

3.1 Data mapping

The initial data mapping, as specified by the application, can remain static or be altered by the application throughout the execution. For dense linear algebra, the two-dimensional block-cyclic data distribution [39] provides a good example of static mapping, shown in Algorithm 6.

When the data mapping is altered by the application, the new data mapping information must be submitted on every node at the same logical point in the sequential task submission flow. The model indeed requires all nodes to have the same knowledge of the data distribution, for Sends and Receives to be posted coherently. However, the data mapping information is

Algorithm 6 Specifying a two-dimensional block-cyclic data distribution

```
for (m = 0; m < N; m++) do
  for (n = 0; n < N; n++) do
    starpu_data_register(&Ahandles[m][n], A[m][n], ...);
    starpu_data_set_rank(Ahandles[m][n], m%P+(n%Q)*P);
  end for
end for
```

only used at *submission* time, to determine which MPI send and receive requests to initiate to resolve dependencies arising from tasks being submitted. Thus, such a redistribution step is not a problem for performance: computing a new data redistribution can be done asynchronously while the previously submitted tasks are still being scheduled and executed. Once determined, the new data redistribution is enforced asynchronously and transparently, by initiating MPI transfers in the background. Meanwhile, subsequent task submissions now proceed using the new data distribution.

This allows to pipeline the application: a part of the graph is submitted, some of it is executed, statistics can be collected on it, a new data distribution can be computed accordingly while the submitted tasks continue executing, and another part of the graph can be submitted with the new distribution, etc. For very large systems, a global redistribution would be very costly, possibly beyond the time to process the already-submitted tasks. Local redistribution schemes could be used instead in this case.

3.2 Task mapping

The task mapping can be made static too, as specified by the application. However, it is usually more convenient to let it automatically determined by StarPU from the data mapping, that is, let tasks move to data. Thus, by default, StarPU will execute a task on the node which owns the data to be written to by that task. Other automatic heuristics can be used to avoid having to specify both a data mapping and a task mapping.

3.3 Discussion

All in all, by inferring the MPI transfers automatically from data dependence edges, separating the data distribution from the algorithm, and inferring task mapping automatically from the data distribution, the *distributed* version of tile Cholesky boils down to Algorithm 7. The only difference with Algorithm 3 in the main loop is the usage of the *starpu_mpi_task_insert* directive instead of `starpu_task_insert`. This function call determines if communications have to be posted through the StarPU-MPI layer (described in Section 2.4) additionally to the baseline StarPU actual computational task submission. The optional `starpu_mpi_cache_flush` directive can be ignored for the moment, we will motivate its usage in Section 4.3.2.

The overall algorithm is sequential and extremely compact ensuring hence very productive development and maintenance processes: algorithms can be designed and debugged within a sequential context, and parallelism can be safely enabled afterwards with the runtime system, transparently for the programmer.

Algorithm 7 Distributed STF Tile Cholesky

```
for (k = 0; k < N; k++) do
  starpu_mpi_task_insert(&POTRF_cl,
      STARPU_RW, A[k][k], 0);
  for (m = k+1; m < N; m++) do
    starpu_mpi_task_insert(&TRSM_cl,
        STARPU_R, A[k][k],
        STARPU_RW, A[m][k], 0);
  end for
  starpu_mpi_cache_flush(A[k][k]); /* See Section 4.3.2 */
  for (n = k+1; n < N; n++) do
    starpu_mpi_task_insert(&SYRK_cl,
        STARPU_R, A[n][k],
        STARPU_RW, A[n][n], 0);
    for (m = n+1; m < N; m++) do
      starpu_mpi_task_insert(&GEMM_cl,
          STARPU_R, A[m][k],
          STARPU_R, A[n][k],
          STARPU_RW, A[m][n], 0);
    end for
    starpu_mpi_cache_flush(A[n][k]); /* See Section 4.3.2 */
  end for
end for
```

4 Experiments

As discussed above, the STF model removes the programming burden of explicitly posting the communications (requested with the MPI paradigm) or explicitly providing the dependencies (requested with the PTG paradigm) as it infers them automatically from the data mapping. We now show that it is possible to rely on this sequential task-based paradigm while achieving high performance on supercomputers. After describing the experimental context (Section 4.1), we first present the final results we obtained (Section 4.2). We eventually list the major issues we faced together with the solutions we have devised to cope with them (Section 4.3) in order to ensure an overall extremely efficient support for the STF model on modern distributed memory machines. We focused on the particular case of a dense linear algebra algorithm to illustrate our study because DPLASMA/PaRSEC is a solid reference in terms of scalability of high performance numerical libraries and was also the challenged code in [22].

4.1 Experimental Context

We implemented the tile Cholesky factorization proposed in Algorithm 7 within the Chameleon library, extending [4] to handle distributed memory machines. We compare the resulting code with two state-of-the-art distributed linear algebra libraries: DPLASMA and ScaLAPACK. DPLASMA implements a highly optimized PTG version of the tile Cholesky factorization introduced in Algorithm 2 on top of the PaRSEC runtime system. ScaLAPACK is the MPI state-of-the-art reference dense linear algebra library for homogeneous clusters. All three libraries rely on the same two-dimensional block-cyclic data distribution [39] between MPI nodes. All the experiments were conducted in real double precision arithmetic.

We conducted our experiments on up to 144 heterogeneous nodes of the TERA100 heterogeneous cluster [40] located at CEA, France. The architectural setup of the nodes and the libraries we used is as follows:

- CPU QuadCore Intel Xeon E5620 @ 2.4 GHz x 2,
- GPU NVIDIA Tesla M2090 (6 GiB) x 2,
- 24 GiB main memory,
- Infiniband QDR @ 40 Gb/s,
- BullxMPI 1.2.8.2,
- Intel MKL 14.0.3.174,
- NVIDIA CUDA 4.2.

For our experiments, we distinguish two setups depending on whether GPUs are used for computation or not. The former will be called the *heterogeneous* setup, while the latter will be called the *homogeneous* setup. We tuned the block size used by each linear algebra library and the schedulers used by each runtime system for each setup, as shown in Table 1, in order to get the best asymptotic performance for each library. In the heterogeneous case, note that DPLASMA is able to efficiently exploit a lower tile size (320) than Chameleon (512) as it supports multi-streaming (see below). PaRSEC and StarPU respectively rely on their own Priority Based Queue (*PBQ*) and priority (*prio*) schedulers, which are both hierarchical queue-based priority scheduler implementations. In the heterogeneous case, StarPU relies on the *dmdas* scheduler, a dynamic variant of the HEFT algorithm, and PaRSEC relies on an extension of PBQ allowing for greedy GPU offloading [41].

Table 1: Tuned parameters for each linear algebra library and for each setup. The block size represents the tile size in the Chameleon and DPLASMA cases and the panel width in the ScaLAPACK case.

Model/Library	CPU-only		CPU + GPU	
	Block size	Sched.	Block size	Sched.
STF/Chameleon	320	prio	512	dmdas
PTG/DPLASMA	320	PBQ	320	PBQ
MPI/ScaLAPACK	64	static	-	-

4.2 Final results

We present in this section the final performance results obtained in our study. Table 2 lists the implementation-specific parameters which were selected during the experiments, to enhance performance to the best of each runtime capability. The main differences are the following:

- While StarPU can execute any kernel of the Cholesky factorization on GPUs, PaRSEC chooses to offload only GEMM kernels on GPUs since this kernel is the most compute intensive one. PaRSEC has support for multi-streaming on GPUs, which allows to execute several kernels at the same time on a single GPU.

- Regarding the communications, the STF model supported by StarPU with Algorithm 7 infers point-to-point communications (we discuss in Section 4.3.2 how to alleviate their repetitions). The PTG model supported by PaRSEC with Algorithm 2 allows for collective communication patterns.

- StarPU takes advantage of the memory pinning optimization of the OpenMPI library to accelerate communications, while it has been turned off with PaRSEC as it induced notable slowdowns.

Table 2: Implementation-specific parameters for each runtime system.

Model/Library (runtime)	Schedulable kernels on GPUs	GPU multi-streaming	MPI comm. policy	OpenMPI memory pinning
STF/Chameleon (StarPU)	All	No	Point-to-point	Yes
PTG/DPLASMA (PaRSEC)	GEMM only	Yes	Collectives	No

Figure 2 shows the performance results obtained on all 144 nodes (1152 CPU cores and 288 GPUs in total) both in the homogeneous (only CPUs being used) and the heterogeneous (all computational units being used) cases. The main observation is that the STF model is competitive with the PTG and the MPI programming paradigms. This is also the main message of the present paper as it shows that very high performance can be reached with the high level of productivity delivered by this model. Furthermore, the observed differences in terms of performance between Chameleon (which implements the STF model) and DPLASMA (which implements the PTG model) are mostly due to the inherent low-level features of the underneath runtime systems (StarPU and PaRSEC, respectively) discussed above, but not to the respective programming models.

In the homogeneous setup, Chameleon and DPLASMA surpass ScaLAPACK. They fully exploit the tiling of the Cholesky factorization. Chameleon and DPLASMA achieve the same asymptotic performance for which both codes have been tuned. On small matrices, DPLASMA slightly outperforms Chameleon. Indeed, StarPU is optimized for heterogeneous architectures while PaRSEC has been originally designed to efficiently exploit homogeneous clusters. In this particular case, the *prio* scheduler implemented in StarPU would need to have support for locality as does the PBQ scheduler implemented in PaRSEC.

In the heterogeneous setup, Chameleon outperforms DPLASMA up to matrices of order 320,000. The OpenMPI memory pinning greatly accelerates StarPU point-to-point communications, thus unlocking enough parallelism to feed GPUs with computation. Furthermore, the StarPU *HEFT* scheduling policy allows for better decisions on the starting phase of the factorization, while the PaRSEC *greedy* policy enforces all initial updates on the GPU despite the cost of transferring the data. For larger matrices, both StarPU and DPLASMA achieve a performance close to the GEMM peak (computed as the sum of the GEMM CPU core and GPU performance over all computational units), with a slight advantage for DPLASMA due to the multi-streaming support of PaRSEC which allows for a better trade-off between parallelism and granularity.

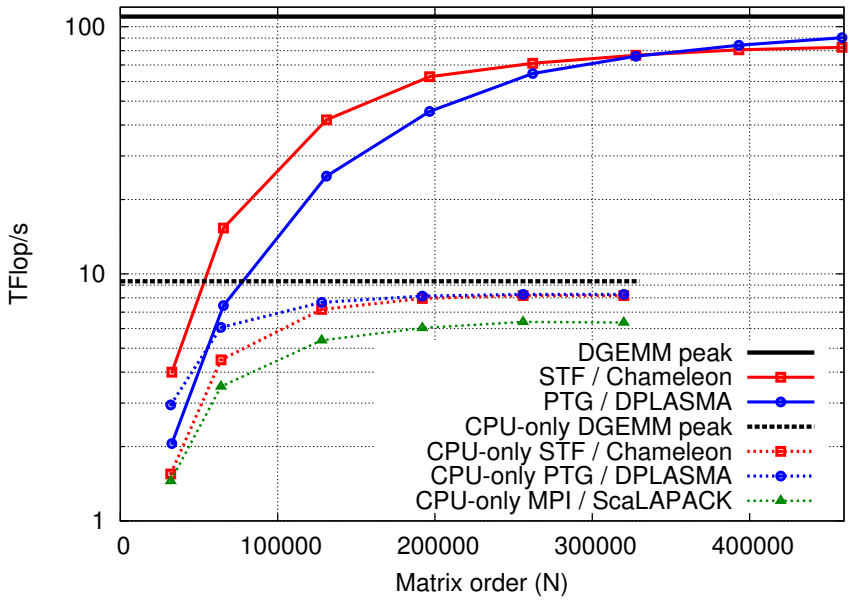


Figure 2: Final performance results on all 144 nodes (1152 CPU cores and 288 GPUs in total). The panel width for ScaLAPACK is 64. The tile sizes for DPLASMA and Chameleon are 320 in the CPU-only case. In the heterogeneous case, they are equal to 320 and 512 for DPLASMA and Chameleon, respectively. The Y-axis is a logarithmic scale.

4.3 List of optimizations required to efficiently support the STF model on supercomputers

We have shown above that we could successfully achieve competitive performance with the STF programming model on modern supercomputers against reference linear algebra libraries implemented with the PTG and MPI paradigms. We now present issues we have faced during this study and how we solved them in order to achieve this performance on those distributed memory machines.

The first issue relates with the STF model inherent requirement, and potential overhead, to submit tasks sequentially for the whole, distributed task graph. We explain in Section 4.3.1 that the characteristics of the global task graph make it suitable for drastic pruning at the node level, thus preserving scalability on large platforms.

The second issue is that inter-node dependence edges trigger duplicate, redundant network data transfers. We discuss in Section 4.3.2 the communication cache mechanism we implemented to filter out redundant network transfers.

The third issue is that the common memory allocator (such as provided by the locally available C library) incurs either critical overhead or fragmentation. In Section 4.3.3, we study the characteristics of the two memory related system calls provided by Linux and similar operating systems, and how these characteristics impair the C library memory allocator, and in the end, how StarPU gets impaired as well. We then present our solution to this issue, involving the cooperation of a memory allocation cache mechanism, together with a task submission flow control policy further detailed in Section 4.3.4.

4.3.1 Pruning of the Task Graph Traversal

The distributed STF paradigm specifies that the full unrolling of the task graph must be done on all participating nodes, even if only a sub-part of the task graph is actually executed on a given node. Such a requirement could hinder scalability. In reality however, a given node only needs to unroll the incoming, local and outgoing dependence edges from its own point of view. Thus, a node may safely prune a task submission for which it is not an end of any of the task's data dependence edges.

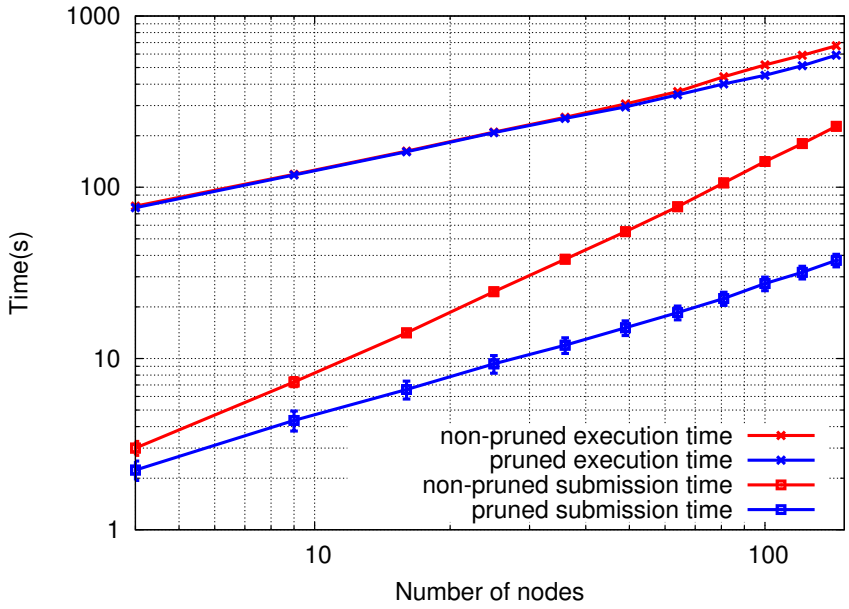


Figure 3: Impact of pruning the task graph on submission and execution time. The test case for 1 node is a matrix of size 40,960, and we keep the same amount of memory per node when increasing the number of nodes.

Figure 3 presents a comparative study of task submission time and execution time depending on whether the application pruned the task graph traversal or not. We first observe that the slope of the execution time without pruning slightly increases beyond 64 nodes, as the resulting of the steep, non-pruned submission time curve reaching the same order of magnitude as the execution time. The pruned submission time curve shows a much more gentle slope instead. If we extrapolate those curves with linear estimations, the extrapolated submission and execution time lines cross at 1,000 nodes without pruning, and at 1,000,000 nodes with pruning. Thus, it convinces that pruning the task graph lowers the submission cost enough to be able to scale up to much more GPU-accelerated computing nodes than the size of the heterogeneous cluster we used (144 nodes).

4.3.2 Communication Cache: Limiting Communications and Memory Footprint

Our distributed model unrolling the (pruned) task graph on all nodes is a fully distributed process, inherently more scalable than a master-slave model by design. Inter-node communications are initiated from local, decoupled decision on each side, upon encountering the node's incoming

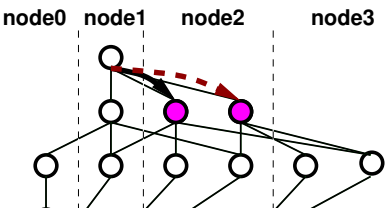


Figure 4: An example of duplicated communications.

and outgoing dependence edges during the DAG traversal, instead of using explicit synchronization requests between nodes. Without additional measures however, multiple dependence edges involving the same pair of nodes and the same piece of data could trigger redundant network transfers if the corresponding piece of data has not been modified in-between. Figure 4 shows the typical pattern of such a redundant communication. We thus filter redundant messages out using a communication cache mechanism to keep track of the remote data copies that a node already received and which are still up-to-date. The sequential submission of the task graph ensures that the communication cache system follows the expected sequential consistency. In particular, it invalidates a cache entry whenever a remote task writing to the corresponding piece of data is encountered during the task graph traversal.

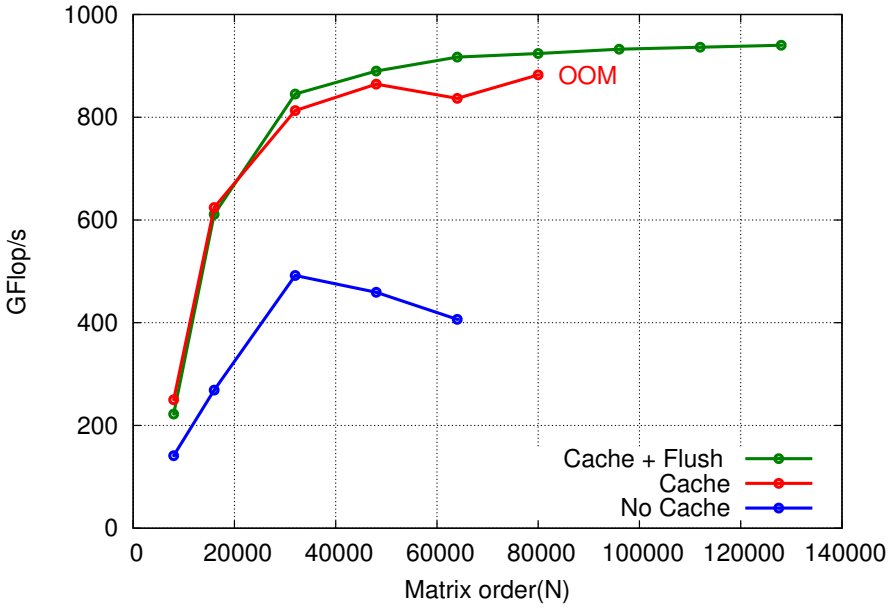


Figure 5: Impact of communication cache policy on performance.

Figure 5 shows performance results depending on the cache policy, on 16 nodes of the homogeneous setup. Comparing the 'Cache' and 'No Cache' policies confirms the impact on performance of filtering redundant network transfers out through caching. However, the 'Cache' policy reaches an out-of-memory condition for matrices larger than 80,000. With the exception of pieces of data written into, the runtime system indeed cannot decide when a cached remote copy can safely be evicted, without additional information from the application. All valid copies are therefore kept

in cache by default.

Hence, we added a method to the StarPU API to allow the application to notify StarPU when a piece of data can safely be flushed from the communication cache. StarPU will actually flush the corresponding cache entry once all tasks using it and submitted before the flush have been executed. Beyond that point, if StarPU encounters a task referencing this piece of data, a new network request will be triggered to get a valid copy. This flush operation can be seen as a notification inserted at some point of the STF submission flow to inform StarPU that the piece of data is possibly not needed by subsequent tasks in a near future. Algorithm 7 shows how this can be added quite naturally within the Cholesky factorization, to express for instance, that the result of a POTRF task is not used beyond the corresponding TRSM tasks. It should be noticed that this is only a communication optimization, which does not change any semantics. The application developer can thus insert and remove them without altering the computation correctness. The 'Cache + Flush' policy curve of Figure 5 confirms that the memory footprint can be greatly reduced this way, and that the performance peak can be sustained for matrices of a much larger scale.

4.3.3 Runtime-level Allocation Cache

The Linux operating system, as many other UNIX-like systems do, provides the two system calls *sbrk* and *mmap* for user-level processes to allocate memory pages. These system calls are then used to build user-land memory allocators such as provided through the *malloc* routine from the C language standard library. The *sbrk* system call allocates (resp. frees) memory pages by increasing (resp. decreasing) the upper limit of the process heap. The *mmap* system call allocates (resp. frees) memory pages virtually anywhere in the addressing space range between the process heap upper limit and the process stack lower limit. The memory allocator implemented by the GNU C Library (glibc) that we used in our study can use both system calls. A threshold on a memory allocation request size dictates whether *sbrk* will be used (below that threshold) or *mmap* will be used (above that threshold). The threshold can be changed by the user program, which also permits to disable one policy or the other entirely.

The two systems calls incur different side effects from the glibc memory allocator. For malloc requests handled through *mmap*, every *malloc* routine call results in a *mmap* system call and every *free* call results in a corresponding *munmap*. The *munmap* operation is expensive since it requires flushing TLB entries, which is notably costly on multicore and/multiprocessor systems.

For *malloc* requests served using the process heap, the *sbrk* system call is called only when needed, that is when the heap is full, or when the heap does not have a contiguous region of free memory large enough for the *malloc* request. This may allow to save on expensive system calls but is prone to fragmentation. Indeed, depending on the ordering of allocation requests and deallocation requests, a large gap resulting from freeing an object (a matrix tile for instance, in our case) may get partially filled by subsequent small allocations (such as allocating StarPU's internal structures); the shortened gap may then not be large enough to accommodate a new object (a new tile for instance) allocation. In that case, the memory allocator is forced to allocate fresh memory pages by increasing the heap upper bound with a call to *sbrk*. Moreover, the allocator may call *sbrk* to decrease the heap upper bound only when a large enough and contiguous chunk of memory is freed at the top of the heap. Here again, the ordering of allocations and frees may prevent such a sufficiently large contiguous memory chunk to be freed at the top of the heap, preventing memory reuse and contributing to increase the overall memory footprint. When the allocator serves requests through *mmap*, the 1-to-1 relationship between *malloc*/*mmap* and between *free*/*munmap* prevents such fragmentation and memory footprint overhead issues, at the cost of more frequent expensive system calls.

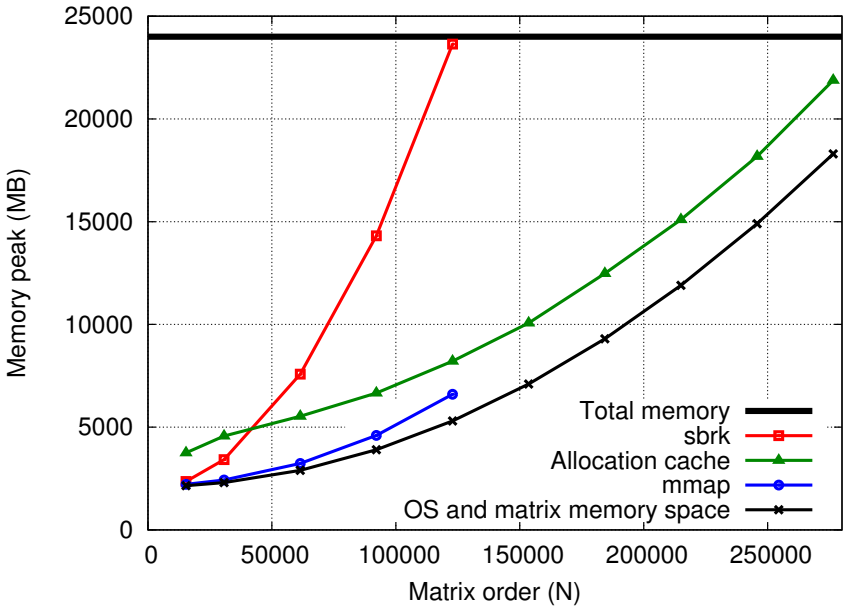


Figure 6: Memory footprint with each allocator and the allocation cache system.

Figures 6 and 7 show the memory footprint and the performance of our Cholesky factorization respectively, on 36 nodes with the homogeneous setup for both the *sbrk*-based and the *mmap*-based allocating policies. Figure 6 shows the memory consumption peak of the most loaded node during the execution.

For the *sbrk*-based policy, our tests give strong evidences of both the fragmentation issue and the issue of the inability to shrink the heap (which often come together). Indeed, when a matrix tile is freed, the *sbrk*-based policy will then try to fill that zone with other allocations such as task structures, instead of preserving that empty space to be reused for another matrix tile. Moreover, since the heap never shrinks, the memory freed at the application level never gets returned to the operating system. These issues result in the steep memory footprint curve of the *sbrk*-based policy on Figure 6. The *mmap*-based policy, which does not suffer from these issues, shows a much better behavior in terms of memory footprint on Figure 6. However, the expensive cost of the system calls it incurs leads to performances deteriorating quickly and dramatically as shown on Figure 7. We thus would like to combine the *reuse* benefits of the *sbrk*-based policy to offset system call costs, and the *mmap*-based policy benefits of being oblivious to fragmentation.

We therefore developed and integrated an allocation cache mechanism in StarPU. The mechanism is built on top of *mmap*, in order to be practically immune to fragmentation. It implements pools of reusable memory chunks grouped by size, which drastically reduces the number of expensive calls to `mmap` and `munmap` by recycling memory areas from completed tasks —StarPU internal data structures, user data, flushed networking buffers— for newly submitted tasks. The results presented on Figures 6 and 7 using the allocation cache (together with task submission control flow, see Section 4.3.4) confirm it enables to combine the best of both worlds, with the added advantage of a much better scalability, while the *sbrk*-based policy quickly leads to memory exhaustion and the *mmap*-based policy quickly leads to execution time explosion.

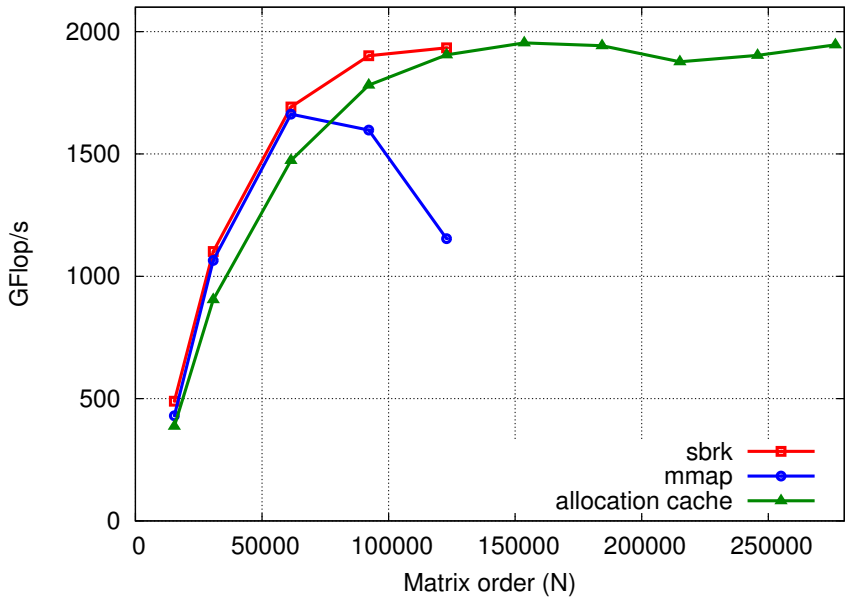


Figure 7: Performance with each allocator and the allocation cache system.

4.3.4 Controlling the task submission flow

Submitting tasks as soon as possible allows the runtime system to take decisions early, such as to infer and post the inter-node communications sufficiently ahead of time, to enable (for instance) efficient computation/communication overlapping. However, unconstrained bulk submissions of tasks may also lead to unwanted side effects.

The effectiveness of the allocation cache (see Section 4.3.3) is directly dependent on the opportunity, by new tasks being submitted, to reuse memory areas allocated by older tasks that have since gone to completion. Without any constraint on the task submission flow, this reuse opportunity hardly ever happens: the submission time per task is usually much shorter than the execution time (as shown in Figure 3), thus all tasks may already have been submitted by the time opportunities for memory reuse starts to arise from task completions.

This problem is emphasized in the case of a distributed session, since StarPU must allocate a buffer for each receive network request posted as the result of submitting a task with an incoming remote dependence. This enables overlapping memory communications with computations. However, this also can lead to the premature allocation of numerous buffers well in advance of their actual usefulness, without additional precautions. The consequences are a larger memory footprint and fewer opportunities for memory reuse, due to a larger subset of the buffers being allocated at overlapping times. Here again, the main reason for this issue is that the submission task front usually runs largely ahead of the execution task front.

A throttling mechanism is therefore necessary on the task submission flow, to keep the memory footprint under control by making the allocation cache effective and by preventing massive, premature allocations of networking buffers. We therefore extended the StarPU API to provide a method for the application to "voluntarily" wait for the number of tasks in the submitted queue to fall below some threshold before continuing its execution. This method can be called for instance at the beginning of an external loop in the application task submission loop nest,

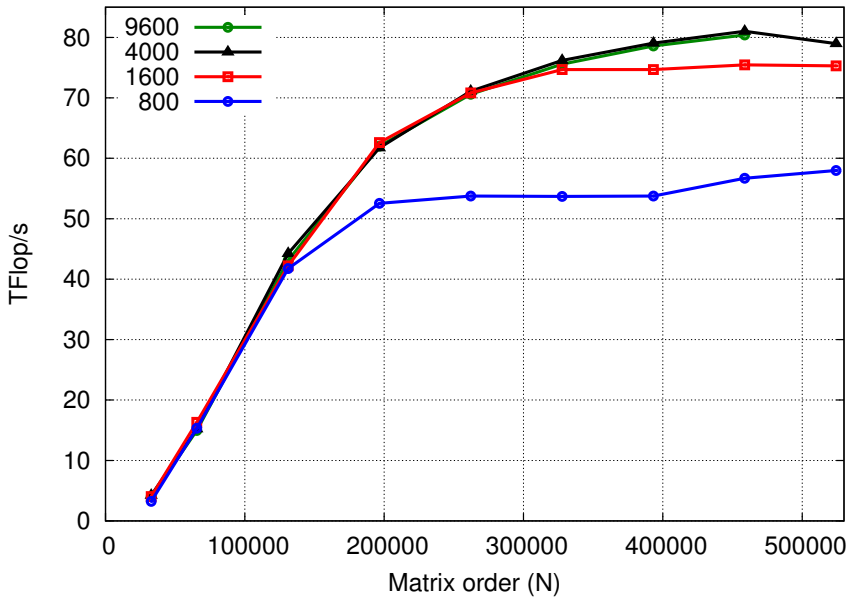


Figure 8: Performance for several number of tasks thresholds.

to wait for some previously submitted phases to progress, before resuming to submit some new phases. The STF property guarantees that the execution cannot deadlock as the result of task submission being temporarily paused.

For some applications, inserting the voluntarily wait method is not practical or desirable. We hence also provide a similar, but transparent, mechanism at the task submission level inside StarPU. Two environment variables allow to specify an upper and a lower submitted task threshold. When the number of task in the submitted task queue reaches the upper threshold upon a new task submit, the task submission method becomes temporarily blocking. It blocks until the number of remaining tasks in the submitted queue falls below the lower threshold upon which task submission resumes.

Figures 8 and 9 present the performance and memory footprint of our application when the task submission flow is voluntarily controlled at application level, for several choices of submitted task thresholds. Before submitting a new phase of tasks, our application waits until the number of tasks remaining in the submission queue falls below that threshold. Both figures show that a classical trade-off has to be made between available parallelism and lookahead depth on the one side, and memory footprint on the other side, when choosing the value of the threshold. Indeed, we observe that runs using a low task threshold result in lower memory footprint but perform worse due to the more limited available parallelism. Conversely, runs using a high task threshold perform better but have a larger memory footprint, due to the additional networking buffers to allocate simultaneously for an increased number of pending incoming requests.

We explored other criteria for the task submission throttling. One of them is a memory footprint criterion, which temporarily pauses task submission when the amount of memory in use reaches the available size of the system. This ensures that applications with datasets larger than the available memory on the machine may still complete successfully. This was implemented in the StarPU runtime system, and experimented as further work [42], notably using an application with unpredictable memory footprint.

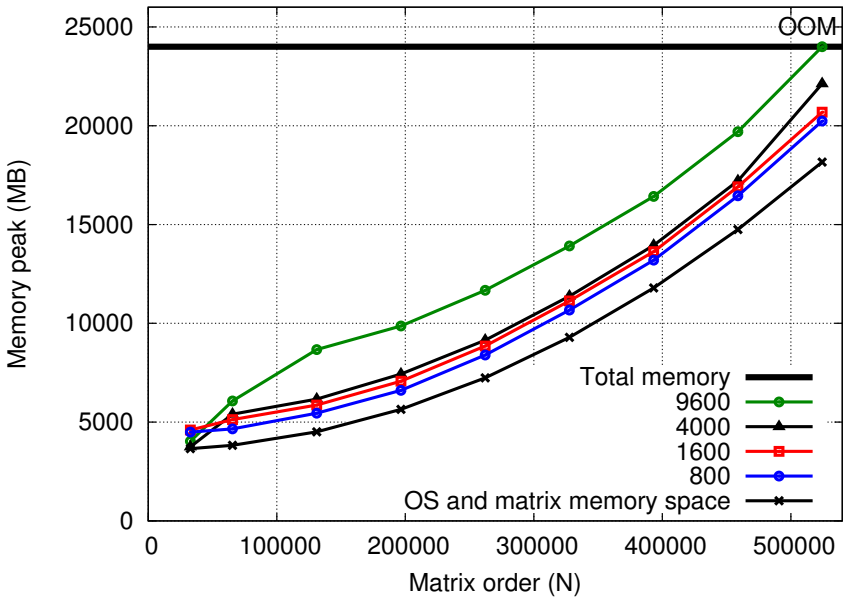


Figure 9: Memory footprint for several number of tasks thresholds.

5 Conclusion and perspectives

Distributed memory machines composed of multicore nodes possibly enhanced with accelerators have long been programmed with relatively complex paradigms for efficiently exploiting all their resources. Nowadays, most of the HPC applications indeed post communication requests explicitly (MPI paradigm) and their most advanced versions even rely on hybrid programming paradigms in order to better cope with multicore chips (such as MPI+OpenMP) and possibly exploit their accelerators (MPI+OpenMP+Cuda). Task-based paradigms aim at alleviating this programming burden. The PTG paradigm was designed to achieve a high scalability. It encodes a DAG of tasks as a data flow. However it requires one to explicitly encode the dependencies between tasks. The STF model removes the programming burden of explicitly posting the communications or explicitly providing the dependencies as it infers them automatically. [21] proposed a support of the STF model on homogeneous clusters using a master-slave model. To achieve a higher scalability, YarKhan [22] proposed to unroll the DAG concurrently on all processing units. Yet, the conclusion of the thesis was that "since [PaRSEC, which implements the PTG model] avoids the overheads implied by the [task] insertion [...] it achieves better performance and scalability [than the proposed extension of quark, which supports the STF model]". In our study, we showed that the STF model can actually compete with the PTG model and we presented the key points that need to be implemented within a runtime system that supports this model to ensure a competitive scalability. Furthermore, our resulting model fully abstracts the architecture and can run any task on any computational unit, making it possible to devise advanced scheduling policies that can strongly enhance the overall performance on heterogeneous architectures as we showed. All in all, we could achieve very high performance on a heterogeneous supercomputer while preserving the fundamental property of the model that a unique sequence of tasks is submitted on every node by the application code, to be asynchronously executed in parallel on homogeneous and heterogeneous clusters. To prove this claim, we have extended the

StarPU runtime system with an advanced inter-node data management layer that supports this model by automatically posting communication requests. Thanks to this mechanism, an existing StarPU STF application can be extended to work on clusters, merely by annotating each piece of data with its node location to provide the initial data distribution, independently from the algorithm itself. From this data distribution, StarPU infers both the task distribution, along the principle that tasks run on the node owning data they write to, and the inter-node data dependencies, transparently triggering non-blocking `MPI_Isend`/`MPI_Irecv` as needed.

We discussed our design choices and techniques ensure the scalability of the STF model on distributed memory machines. Following [22], we made the choice of a fully decentralized design, made possible by the STF paradigm: every node-local scheduler receives the same task flow from the application, and thus gets a coherent view of the global distributed state, without having to exchange explicit synchronization messages with other nodes. While submitting the whole graph on every node could raise concerns about the scalability [22], we showed that the flow of tasks actually submitted on a given node can be drastically simplified to its distance-1 neighborhood, constituted from the tasks having a direct incoming and/or outgoing dependence with the tasks of this given node. We furthermore implemented two cache mechanisms to offset the expensive cost of memory allocations and avoid redundant data transfer requests, namely the allocation cache and the communication cache respectively. Finally, we designed a throttling mechanism on task submission flow, to monitor resource subscription generated by the queued tasks, and to cap the task submission rate in accordance. Combining the StarPU-MPI layer with these optimizations enabled achieving high performance with the Cholesky factorization on a heterogeneous supercomputer. We showed our approach to be competitive with the state-of-art DPLASMA and ScaLAPACK libraries. All the software elements introduced in this paper are available as part of the StarPU runtime system and the Chameleon dense linear algebra libraries.

On-going and future work on the distributed STF support in StarPU will mainly focus on extending the automation capabilities, on integrating a distributed load balancing engine and generalizing the StarPU-MPI layer networking support. We intend to extend the automation capabilities of StarPU to provide the application programmer with sensible auto-determined thresholds, in particular for the task submission capping mechanism. Indeed, since the choice of this parameter value is a trade-off between parallelism and memory footprint, we would like to monitor the memory footprint of tasks, so that tasks can be submitted until all the memory allowed to be used by the runtime system is subscribed. We plan to integrate a load balancing mechanism to alter the initial data distribution automatically during the execution to even out the dynamically observed computational load imbalance on every node, relieving the application from that burden. Finally, we are porting the abstract part of StarPU distributed support on new networking interfaces beyond MPI.

While the STF model has been supported in the OpenMP standard since the 4.0 revision for shared-memory machines with the introduction of the *depend* clause, we expect that the present work will open up new perspectives for OpenMP towards the support of distributed memory machines.

Acknowledgments

This work was done with the support of ANR SOLHAR (grant ANR-13-MONU-0007) as well as the PlaFRIM and Curie/CCRT computing centers. The authors furthermore thank Cédric Augonnet, David Goudin and Xavier Lacoste from CEA CESTA who strongly contributed to this achievement as well as Luc Giraud and Emmanuel Jeannot for their feedback on a preliminary version of the manuscript.

References

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[2] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, J. Kurzak, P. Luszczek, S. Tomov, , and J. Dongarra, "Scalable dense linear algebra on heterogeneous hardware," pp. 65–103, 2013.

[3] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. Van De Geijn, "FLAME: Formal linear algebra methods environment," *ACM Transactions on Mathematical Software (TOMS)*, vol. 27, no. 4, pp. 422–455, 2001.

[4] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A hybridization methodology for high-performance linear algebra software for GPUs," *in GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484, 2011.

[5] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. Phoenix, United States: IEEE, May 2014, pp. 29–38.

[6] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems," IRIT, Tech. Rep. IRI/RT–2014-03–FR, November 2014, to appear in ACM Transactions On Mathematical Software. [Online]. Available: http://buttari.perso.enseeiht.fr/stuff/IRI-RT--2014-03--FR.pdf

[7] E. Agullo, L. Giraud, A. Guermouche, S. Nakov, and J. Roman, "Task-based conjugate gradient: from multi-GPU towards heterogeneous architectures," Tech. Rep., 2016.

[8] H. Ltaief and R. Yokota, "Data-Driven Execution of Fast Multipole Methods," *CoRR*, vol. abs/1203.0889, 2012.

[9] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[10] B. Lizé, "Résolution directe rapide pour les eléments finis de frontiere en electromagnétisme et acoustique: $\mathcal{H}$-matrices. Parallélisme et applications industrielles," Ph.D. dissertation, Paris 13, 2014.

[11] T. Heller, H. Kaiser, and K. Iglberger, "Application of the parallex execution model to stencil-based problems," in *International Supercomputing Conference*, 2012.

[12] L. Boillot, G. Bosilca, E. Agullo, and H. Calandra, "Task-based programming for Seismic Imaging: Preliminary Results," in *2014 IEEE International Conference on High Performance Computing and Communications (HPCC)*. Paris, France: IEEE, Aug. 2014, pp. 1259–1266. [Online]. Available: https://hal.inria.fr/hal-01057580

[13] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in *The 1990 ACM/IEEE conference on Supercomputing*, 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=110382.110385

[14] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs," in *GPU Computing Gems*, Wen-mei W. Hwu, Ed., 2010. [Online]. Available: http://hal.inria.fr/inria-00547847/en/

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, Feb. 2011.

[16] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02, Tech. Rep., 2011.

[17] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, Dec. 2010. [Online]. Available: http://hal.inria.fr/inria-00523937

[18] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 2. IEEE, 1995, pp. 113–122.

[19] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. J. Dongarra, "Distributed Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA," in *Proceedings of the 25th IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, Anchorage, United States, May 2011, pp. 1432–1441. [Online]. Available: https://hal.inria.fr/hal-00809680

[20] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[21] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "A high-productivity task-based programming model for clusters," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 18, pp. 2421–2448, 2012. [Online]. Available: http://dx.doi.org/10.1002/cpe.2831

[22] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tennessee, 2012.

[23] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *International conference on Supercomputing*, 2013.

[24] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed {DAG} engine for high performance computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37 – 51, 2012, extensions for Next-Generation Parallel Programming Models. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347

[25] M. Tillenius, E. Larsson, E. Lehto, and N. Flyer, "A task parallel implementation of a scattered node stencil-based solver for the shallow water equations," in *Swedish Workshop on Multi-Core Computing*, 2013.

[26] A. Zafari, M. Tillenius, and E. Larsson, "Programming models based on data versioning for dependency-aware task-based parallelisation," in *International Conference on Computational Science and Engineering*, 2012.

[27] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independene with logical regions," in *International conference on high performance computing, networking, storage and analysis*, 2012.

[28] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[29] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312025

[30] T. D. Hartley, E. Saule, and Ümit V. Çatalyürek, "Improving performance of adaptive component-based dataflow middleware," *Parallel Computing*, vol. 38, no. 6–7, pp. 289 – 309, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819112000221

[31] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *The 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[32] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Locality-aware work stealing on multi-CPU and multi-GPU architectures," in *Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.

[33] D. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, University of Illinois, 2012.

[34] D. M. Kunzman and L. V. Kalé, "Programming heterogeneous clusters with accelerators using object-based programming," *Scientific Programming*, 2011.

[35] Y. Zheng, C. Iancu, P. H. Hargrove, S.-J. Min, and K. Yelick, "Extending Unified Parallel C for GPU Computing ," in *SIAM Conference on Parallel Processing for Scientific Computing (SIAMPP)*, 2010.

[36] J. Lee, M. T. Tran, T. Odajima, T. Boku, and M. Sato, " An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters ," in *HeteroPar*, 2011.

[37] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *Proceedings of the International Euro-Par Workshops 2009, HPPC'09*, ser. Lecture Notes in Computer Science, vol. 6043. Delft, The Netherlands: Springer, Aug. 2009, pp. 56–65. [Online]. Available: http://hal.inria.fr/inria-00421333

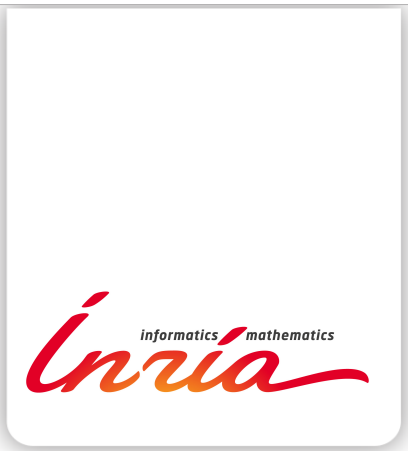
[38] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[39] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, 1997.

[40] "TERA-100 Hybrid," Website, 2015, http://www.top500.org/system/177460.

[41] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical DAG Scheduling for Hybrid Distributed Systems," in *29th IEEE International Parallel & Distributed Processing Symposium*, Hyderabad, India, May 2015. [Online]. Available: https://hal.inria.fr/hal-01078359

[42] M. Sergent, D. Goudin, S. Thibault, and O. Aumage, "Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System," in *21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, United States, May 2016. [Online]. Available: https://hal.inria.fr/hal-01284004



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399